





Estimate the size of B-tree Indexes

	Estimate size of B-tree Indexes	
	Version : 0.3	

VERSION

Version	Date	Description
0.1	03/11/2005	First Edition
0.2	29/11/2005	1. Correction comment on “analyze index index_name validate structure” command. (thanks to Mr Sergio Menoyo)
0.3	27/12/2005	<ol style="list-style-type: none"> 1. Correction comment on relationship ITL & ASSM 2. Improved regarding indexing numeric fields (precision) 3. Local Partitioned Indexes 4. Global Partitioned Indexes 5. Non Partitioned Indexes on Partitioned tables 6. Density of leaf blocks with sys_op_lbid



1. INTRODUCTION.....	4
2. HOW MANY LEAF BLOCKS WILL THERE BE ?.....	5
2.1 OVERHEAD AT THE LEAF BLOCK LEVEL	5
2.2 OVERHEAD PER INDEX ENTRY (ANALYSIS).....	8
2.2.1 <i>The overhead on a varchar2 column</i>	8
2.2.2 <i>The overhead on a char column</i>	8
2.2.3 <i>The overhead on a sequenced primary key column</i>	8
2.2.4 <i>The overhead for numeric datatypes</i>	9
2.2.5 <i>The overhead of a composite index</i>	9
2.2.6 <i>PARTITIONED INDEXES</i>	11
2.2.7 <i>NON PARTITIONED INDEXES ON A PARTITIONED TABLE</i>	14
2.3 OVERHEAD PER INDEX ENTRY (FORMULA)	14
3. TEST	16
4. HOW MANY BRANCH BLOCKS WILL THERE BE ?	17
5. CONCLUSION	18
6. SPARSENESS OF LEAF BLOCKS.....	19
7. REFERENCES.....	21

	Estimate size of B-tree Indexes	
	Version : 0.3	

1. INTRODUCTION

One of our database designers asked me how she should estimate the size of btree indexes. According the documentation there is an overhead per index entry and an overhead per indexed column. Another source speaks from the lock byte of each index entry and a byte in which we store the byte length of the indexed value. Since I found nowhere documentation about the exact sizing of the overhead per index entry neither about the overhead per leaf block and branch block I decided to benchmark it and I feel free to publish my benchmarked results. (benchmarking done with release 10.1 and 10.2)

The btree index consists of leaf blocks, branch blocks and one root block. The root block at the lowest level (level 0) points to branch blocks at level 1, these branch blocks point to other branch blocks at level 2, finally at the highest level we find the leaf blocks. The branch blocks at the highest level - 1 point to these leaf blocks. Each leaf block contains a set of index entries. You can find more info about the index structure in the concept manuals or in Mr Tim Gorman' s "Understanding Indexes" (you can download this excellent white paper at <http://www.evdbt.com>). The views dba_indexes, dba_ind_columns and index_stats are usefull in order to understand, to learn about, to tune the btree structure. Please note index_stats only gets populated by the command "analyze index index_name validate structure" and that this view only contains one row, info of the latest analyzed index. Please note as well that the command "analyze index index_name validate structure" requires a TM lock on the underlying table and although shortly this locking might be unwanted in a production database. If these TM locks are unwanted in a 24x7 system one can use the "analyze index index_name validate structure online" command.

	<p>Estimate size of B-tree Indexes</p>	
	<p>Version : 0.3</p>	

2. HOW MANY LEAF BLOCKS WILL THERE BE ?

Maybe a lot ! Each index entry in a leaf block has got two columns.

- A. the - old Oracle 7 format - 6 byte rowid
- B. the indexed value.

There is an **overhead per index entry**, this overhead consists of

- C. a row header
- D. a column header

There is not only an overhead per index entry, there is as well an **overhead per leaf block (O)**, and this overhead consists of

- O1. Fixed block header (size depends on RDBMS version)
- O2. Variable transaction header (23 bytes per ITL slot)
- O3. The leaf blocks are never fully used.

So the entire formula we are looking for becomes



[(A+vsize(B)+C+D) * n indexed table rows

O1 + O2 + O3

2.1 OVERHEAD AT THE LEAF BLOCK LEVEL

Every new table entry has got it' s index entry. Since the index is sorted -ascending or descending- the new index entry has a specific location, a specific leaf block to go to. If the leaf block is full and a new entry should fit in it ... there will be a **leaf block split** (mostly 50/50) meaning that half of the part of the index entries in that leaf block move to another leaf, meaning the index becomes less dense. New index entries will fill up these 50 pct of free space whereas other leaf blocks might split at that time¹. With the "create index index_name on table_name (column_name) pctfree 20 pct" we can only control at creation time the density of the leaves. With the "alter index index_name rebuild pctfree 20" we can only control at rebuild time the density of the

¹ V\$sysstat has statistics for both leaf and branch node splits

	Estimate size of B-tree Indexes	
Version : 0.3		

leaves. For rather read only systems we can achieve a 90 pct density for the leaves, whereas for OLTP databases a 70 - 75 pct density seems to be more realistic.

Since release 9 Oracle encourage us to implement ASSM tablespaces. With ASSM Oracle has introduced a new free block management. Using the old method of manual segment space management a dba should consider freelists of table segments and pct_used of table blocks as well freelists for index segments. However even with ASSM tablespaces a dba should still configure manually the ITL' s . (with intrans and maxtrans²). According to the Oracle documentation 1 ITL slot takes 23 bytes. If more ITL' s are required then initially created in both table and index blocks Oracle expand dynamically the ITL list assuming there is free space left in the block³. As such we are not exactly sure about the number of ITL' s in an index leaf block. I was dumping both leaf and branch block and found a variable number of ITL' s in it. So O1 + O1 + O3 in the above formula is empiric. I created a couple of indexes on varchars, numbers, chars, a combination of these and I dumped the leaf blocks with the "alter system dump datafile n block n" command. I found back the tracefiles in the userdump directory and however it is out of the scope to document leaf blockdumps. I post here a part of it

Leaf block dump

```



=====
header address 182413412=0xadf6864
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 3
kdxcosdc 0
kdxconro 172 >> These are the number of index entries in this leaf block
kdxcofbo 380=0x17c
kdxcofeo 2020=0x7e4
kdxcoavs 1640
kdxlespl 0
kdxlende 0
kdxlenxt 16860134=0x10143e6 >> This seems to be the next leaf block
kdxleprv 0=0x0
kdxledsz 0
kdxlebsz 8036 >> This seems to be a part from the O1 + O2 ( but not O3 )

```

In order to estimate the number of possible index entries per leaf block I assumed - based on benchmarking- an overhead of 30 pct for OLTP databases (O1+O2+O3=0,7), whereas the overhead might be 20 pct for rather read only systems with an initial index build with let' s say pctfree 10 (O1+O2+O3=0,8). The blocksize of the tablespace in which the index will be created / rebuild is offcourse another parameter. The higher the blocksize the higher factor O.

² Mr Thomas Kyte warned me that maxtrans is obsolete in 10g, it defaults to a single value of 255 (Mr Thomas Kyte has however NOT read this document and as such is NOT responsible for what I write)

³ v\$segment_statistics can be queried in order to list segments suffering from ITL_waits (rather uncommon)

	<p style="text-align: center;">Estimate size of B-tree Indexes</p> <hr/> <p style="text-align: center;">Version : 0.3</p>	
---	--	---

[(A+vsize(B)+C+D) * n indexed table rows



----- (for rather OLTP databases)

0,7

[(A+vsize(B)+C+D) * n indexed table rows

----- (for rather read only databases)

0,8

	<p style="text-align: center;">Estimate size of B-tree Indexes</p> <hr/> <p style="text-align: center;">Version : 0.3</p>	
---	---	---

2.2 OVERHEAD PER INDEX ENTRY (ANALYSIS)

Since I dumped the leaf blocks with the "alter system dump datafile n block n" command I tried to find back the overhead per index entry, I tried to find the C + D.

2.2.1 The overhead on a varchar2 column

This is a part from an index leaf block dump on a varchar(20) column of which the byte length of the indexed value is 3 bytes. (select vsize(indexed_column) from table)

```
row#0[8023] flag: -----, lock: 0, len=13
col 0; len 3; (3): 34 31 30
col 1; len 6; (6): 01 00 10 de 00 0f
```

The col 1 is the 6 byte rowid. The col 0 the indexed value. The overhead per index entry seems to be 4 bytes here. (I kept in mind overhead on a varchar : 4 bytes)

2.2.2 The overhead on a char column

This is a part from an index leaf block dump on a char(20) column of which the byte length of the indexed value is 20 bytes. (select vsize(indexed_column) from table)

```
row#191[2276] flag: -----, lock: 0, len=30
col 0; len 20; (20): 30 33 32 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
col 1; len 6; (6): 01 00 07 62 00 0d
```

The col 1 is the 6 byte rowid. The col 0 the indexed value. The overhead per index entry seems to be 4 bytes here. (I kept in mind overhead on a char : 4 bytes)



2.2.3 The overhead on a sequenced primary key column

This is a part from a sequenced primary key index dump. The vsize(pk) is either 2 either 3 bytes (depending on the value of the numeric field off course).

```
row#0[8001] flag: -----, lock: 0, len=11, data:(6): 01 00 77 27 00 00
col 0; len 2; (2): c1 4d

row#45[7482] flag: -----, lock: 0, len=12, data:(6): 01 00 77 28 00 03
col 0; len 3; (3): c2 0f 22
```

Somewhat different ordered. But again $12 - 3 - 6 = 11 - 2 - 6 = 3$. (I kept in mind overhead on a number : 3 bytes)

	<p>Estimate size of B-tree Indexes</p>	
<p>Version : 0.3</p>		

2.2.4 The overhead for numeric datatypes

2.2.4.1 The overhead on a unique numeric value column (not primary key) number(10)

This is a part from a unique index numeric value dump. The vsize(unique numeric value) is 2 bytes (depends on the value of the numeric field off course).

```
row#0[8001] flag: -----, lock: 0, len=11, data:(6): 01 00 77 27 00 00
col 0; len 2; (2): c1 17
```

Somewhat different ordered. But again $11 - 2 - 6 = 3$. (I kept in mind overhead on a number(10) : 3 bytes)

2.2.4.2 The overhead on a unique numeric value column (not primary key) number(10,2)

I created a empty copy of the table on which I created my number(10) index. Then I modified the datatype, instead of number(10) I made it number(10,2), and this is what the dump us show.

```
row#423[2528] flag: -----, lock: 0, len=13
col 0; len 3; (3): c2 19 27
col 1; len 6; (6): 01 00 36 c3 00 10
```

Again col1 represents the rowid whereas col0 represents the indexed value. Hence we see an overhead of 4 bytes. **The precision of the numeric field is apparently a factor.** (I kept in mind overhead on a number(10,2) : 4 bytes)



2.2.5 The overhead of a composite index

Please note I haven' t benchmarked yet compressed composite indexes as such in this version of my document you cannot use the below findings for compressed indexes.

2.2.5.1 NUMBER + VARCHAR2

This is a part from a composite index leaf block dump, composite : number + varchar(20). The vsize(number_column) is here 4 bytes. The vsize on the varchar(20) column is 3 bytes.

```
row#317[2317] flag: -----, lock: 0, len=18
col 0; len 4; (4): c3 07 19 3a
```

	<p style="text-align: center;">Estimate size of B-tree Indexes</p> <hr/> <p style="text-align: center;">Version : 0.3</p>	
---	--	---

```
col 1; len 3; (3): 37 33 30
col 2; len 6; (6): 01 00 07 c5 00 07
```

The col 2 is the 6 byte rowid. The col 0 the indexed value part number, whereas col 1 is the indexed value part varchar(20) . The overhead per index entry seems to be 5 bytes here. (I kept in mind overhead on a number + varchar = 5 bytes overhead)

2.2.5.2 NUMBER + CHAR

This is a part from a composite index leaf block dump, composite : number + char(20). The vsize(number_column) is here 4 bytes. The vsize on the char(20) column is off course 20 bytes.⁴

```
row#171[2020] flag: -----, lock: 0, len=35
col 0; len 4; (4): c3 07 18 0c
col 1; len 20; (20): 35 33 30 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
col 2; len 6; (6): 01 00 07 96 00 01
```

The col 2 is the 6 byte rowid. The col 0 the indexed value part number, whereas col 1 is the indexed value part char(20) . The overhead per index entry seems to be 5 bytes here. (I kept in mind overhead on a number + char = 5 bytes overhead). Also I tracked down the overhead on a number + varchar = 5 bytes)

2.2.5.3 VARCHAR2 + CHAR

This is a part from a composite index leaf block dump, composite : varchar2(20) + char(20). The vsize(varchar2_column) is here 3 bytes. The vsize on the char(20) column is off course 20 bytes.

```
row#168[2290] flag: -----, lock: 0, len=34
col 0; len 3; (3): 30 33 32
col 1; len 20; (20): 30 33 32 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
col 2; len 6; (6): 01 00 07 48 00 04
```

The col 2 is the 6 byte rowid. The col 0 the indexed value part varchar2, whereas col 1 is the indexed value part char(20) . The overhead per index entry seems to be 5 bytes here. (I kept in mind overhead on a varchar2 + char = 5 bytes overhead)

2.2.5.4 VARCHAR2 + VARCHAR2

This is a part from a composite index leaf block dump, composite : varchar2(20) + varchar2(20). The vsize(varchar2_column) is here 3 bytes for the first column and 14 bytes either 7 bytes on the second column (depending on the indexed value).

```
row#41[7107] flag: -----, lock: 0, len=28
col 0; len 3; (3): 34 31 30
col 1; len 14; (14): 34 30 30 2f 30 30 30 2f 56 34 32 30 32 31
col 2; len 6; (6): 01 00 7b 98 00 12
```

⁴ Be carefull when multibyte character sets comes into play

```
row#28[7387] flag: -----, lock: 0, len=21
col 0; len 3; (3): 34 31 30
col 1; len 7; (7): 34 30 30 2f 30 30 30
col 2; len 6; (6): 01 00 78 87 00 06
```

The col 2 is the 6 byte rowid. The col 0 the indexed value part varchar2, whereas col 1 is the other indexed value part varchar2. The overhead per index entry seems to be 5 bytes here. (I kept in mind overhead on a varchar2 + varchar2 = 5 bytes overhead)

2.2.5.5 VARCHAR2 + VARCHAR2 + CHAR

This is a part from a composite index leaf block dump, composite : varchar2(20) + varchar2(20) + char(20). The vsize(varchar2_column) is here 3 bytes for the first column and 14 bytes whereas the vsize(char(20)) is off course 20 bytes.

```
row#123[1921] flag: -----, lock: 0, len=49
col 0; len 3; (3): 30 33 32
col 1; len 14; (14): 33 30 30 2f 30 30 37 2f 50 30 32 32 32 30
col 2; len 20; (20): 30 33 32 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
col 3; len 6; (6): 01 00 07 6a 00 14
```

The col 3 is the 6 byte rowid. The col 0 the indexed value part varchar2, whereas col 1 is the other indexed value part varchar2 and col2 is the indexed value part char(20). The overhead per index entry seems to be 6 bytes here. (I kept in mind overhead on a varchar2 + varchar2 + char = 6 bytes overhead)

2.2.6 PARTITIONED INDEXES

In the examples below we consider table T_ALERT , partitioned by RANGE and subpartitioned by HASH

```
SQL> desc t_alert;
```



```
ALERT_NB          NUMBER(9)
COUNTRY           CHAR(2 CHAR)
ALERT_TYPE        VARCHAR2(50 CHAR)
ALERT_REASON      VARCHAR2(50 CHAR)
ALERT_DATE        DATE
REASON_TYPE       NUMBER(8,2)
```

```
PARTITION BY RANGE ("REASON_TYPE")
SUBPARTITION BY HASH ("COUNTRY")
SUBPARTITIONS 20
```

```
SQL> select table_name,partition_name,subpartition_count from user_tab_partitions where
table_name = 'T_ALERT';
```

```
TABLE_NAME PARTITION_ SUBPARTITION_COUNT
```

```
-----
T_ALERT     P2          20
T_ALERT     P3          20
T_ALERT     P4          20
```

 10g Certified Professional	Estimate size of B-tree Indexes	
	Version : 0.3	

```
T_ALERT    P5          20
T_ALERT    P6          20
T_ALERT    P7          20
T_ALERT    P8          20
T_ALERT    P9          20
T_ALERT    P10         20
```

2.2.6.1 LOCAL PARTITIONED INDEXES (SINGLE COLUMN)

The local partitioned index has been created on a char(2) column.

```
SQL> select dbms_metadata.get_ddl('INDEX','T_ALERT_IDX3') from dual;

CREATE INDEX "MY_UTF"."T_ALERT_IDX3" ON "MY_UTF"."T_ALERT" ("COUNTRY")
PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE ( BUFFER_POOL DEFAULT) LOCAL (PARTITION "P2"
PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE( BUFFER_POOL DEFAULT)
( SUBPARTITION "SYS_SUBP249" TABLESPACE "USERS",
SUBPARTITION "SYS_SUBP250" TABLESPACE "USERS",
... SUBPARTITION "SYS_SUBP428" TABLESPACE "USERS"))
```

This is a part from a local partitioned index leaf block dump on the table T_ALERT.

```
row#0[8024] flag: -----, lock: 0, len=12
col 0; len 2; (2): 41 4c
col 1; len 6; (6): 01 00 1a 54 00 0c
```



The col 1 is the 6 byte rowid. The col 0 the indexed value part char(2). The overhead per index entry seems to be 4 bytes here. (I kept in mind overhead on a char column = 4 bytes overhead) **Seems no difference compared with non partitioned indexes.**

2.2.6.2 LOCAL PARTITIONED INDEXES (MULTIPLE COLUMNS)

The composite local partitioned index has been created on a char(2) and on a number.

```
SQL> select dbms_metadata.get_ddl('INDEX','T_ALERT_IDX2') from dual;

CREATE INDEX "MY_UTF"."T_ALERT_IDX2" ON "MY_UTF"."T_ALERT" ("COUNTRY",
"REASON_TYPE")
PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE ( BUFFER_POOL DEFAULT) LOCAL
(PARTITION "P2"
PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE ( BUFFER_POOL DEFAULT)
( SUBPARTITION "SYS_SUBP249"
```

	Estimate size of B-tree Indexes	
	Version : 0.3	

This is a part from a composite local partitioned index leaf block dump on the table T_ALERT

```
row#239[8021] flag: -----, lock: 0, len=15
col 0; len 2; (2): c1 02
col 1; len 2; (2): 41 4c
col 2; len 6; (6): 01 00 1a 55 00 93
```

The col 2 is the 6 byte rowid. The col 0 the indexed value part char(2) whereas col 1 is the indexed value part number. The overhead per index entry seems to be 5 bytes here. (I kept in mind overhead on a char column + number column = 5 bytes overhead) **Seems no difference compared with non partitioned indexes.**

Hence we see the overhead of a local partitioned index is similar compared with the overhead for non partitioned indexes on non partitioned tables. (assuming datatype and precision is identical) There are however particular and interesting situations which needs special attention : global partitioned indexes on a partitioned table and non partitioned indexes on a partitioned table.

2.2.6.3 GLOBAL PARTITIONED INDEXES

The global partitioned index has been created on a number(8,2).

```
CREATE INDEX T_ALERT_IDX4 ON T_ALERT(reason_type)
GLOBAL PARTITION BY RANGE (reason_type)
(PARTITION p1 VALUES LESS THAN (2),
PARTITION p2 VALUES LESS THAN (3),
PARTITION p3 VALUES LESS THAN (4),
PARTITION p4 VALUES LESS THAN (5),
PARTITION p5 VALUES LESS THAN (6),
PARTITION p6 VALUES LESS THAN (7),
PARTITION p7 VALUES LESS THAN (8),
PARTITION p8 VALUES LESS THAN (9),
PARTITION p9 VALUES LESS THAN (MAXVALUE));
```

This is a part from a global partitioned index leaf block dump on the table T_ALERT

```
row#398[1652] flag: -----, lock: 0, len=16
col 0; len 2; (2): c1 09
col 1; len 10; (10): 00 03 7b 88 01 00 1e b6 00 06
```

The col 1 is the 6 byte rowid. The col 0 the indexed value part char(2). The overhead per index entry seems to be 4 bytes here. (I kept in mind overhead on a number(8,2) column = 4 bytes).

Hence we see the overhead of a global partitioned index is similar compared with the overhead for non partitioned indexes on non partitioned tables. (assuming datatype and precision is identical) , HOWEVER THE 10 BYTE FORMAT ROWID IS STORED IN THE INDEX LEAF BLOCK !!!

2.2.7 NON PARTITIONED INDEXES ON A PARTITIONED TABLE

I dropped t_alert_idx4 and I created another t_alert_idx5 on the same column, same precision, when we look at our leaf block dumps we see something interesting, the rowid is this time the 10 byte format (format since release 8.03)

```
row#352[2388] flag: -----, lock: 0, len=16
col 0; len 2; (2): c1 02
col 1; len 10; (10): 00 03 7a fc 01 00 1a 55 00 94
```

For non partitioned indexes on partitioned tables we face the same overhead compared with the local/global partitioned indexes on partitioned tables, HOWEVER THE 10 BYTE FORMAT ROWID IS STORED IN THE INDEX LEAF BLOCK !!!⁵

2.3 OVERHEAD PER INDEX ENTRY (FORMULA)

Since my findings didn't permit to be sure of the exact overhead per index entry and per indexed column header I decided to change the C + D to X + 1, as such the empiric formula becomes

[(A + vsize(B) + x + 1) * n indexed table rows

O

X is a variable depending on the datatype and the number of indexed columns.

varchar2 ⇒ x = 4

char ⇒ x = 4

number (without decimal precision) ⇒ x = 3

number (with decimal precision) ⇒ x = 4



varchar2 + varchar2 ⇒ x = 5

varchar2 + char ⇒ x = 5

char + varchar2 ⇒ x = 5

char + char ⇒ x = 5

⁵ Mr Tim Gorman has described this in his "Understanding Indexes"

	Estimate size of B-tree Indexes	
	Version : 0.3	

number (without decimal precision) + char $\Rightarrow x = 5$

number (without decimal precision) + varchar2 $\Rightarrow x = 5$

varchar2 + varchar2 + char $\Rightarrow x = 6$

varchar2 + varchar2 + varchar2 $\Rightarrow x = 6$

char + char + char $\Rightarrow x = 6$

A is a variable depending on the number of bytes of the rowid (6 or 10)

Non partitioned tables $\Rightarrow A = 6$

Local partitioned indexes on partitioned tables $\Rightarrow A = 6$

Global partitioned indexes on partitioned tables $\Rightarrow A = 10$

Non partitioned indexes on partitioned tables $\Rightarrow A = 10$

3. TEST

Assume we have a table eqp_topo with 899.645 entries and assume we have a column eq_zone a varchar2 column with the not null attribute. We wanna create a non unique btree on eq_zone. We now the average indexed value = 3 bytes. (select avg(vsize(eq_zone)) from eqp_topo). Question how big will the index be ?

Let' s create that index in a 8K tablespace

```
SQL> create index ix_eq_zone on eqp_topo (eq_zone) pctfree 20;
Index created.
```

```
SQL> analyze index ix_eq_zone validate structure;6
Index analyzed.
```

```
SQL> select lf_blks*8192,lf_rows,del_lf_rows,pct_used,height from index_stats;
```

```
LF_BKLS*8192 LF_ROWS DEL_LF_ROWS PCT_USED HEIGHT
-----
17.383.424    899645      0           80        3
```

And with the formula, since eq_zone is a varchar x = 4 and the vsize(b) = 3

(6 + 3+ 1+ 4) * 899.645

----- = 17.992.900

0.7

Well that' s not bad , but far from exact science would Isaac Newton have said. (1+1=2 and not 1,98). The btree doesn' t only consist of leaf blocks ... we have branch blocks and 1 root block as well.

⁶ analyze index ix_eq_zone validate structure ONLINE can be used if no TM lock of the underlying table can or may be acquired.

4. HOW MANY BRANCH BLOCKS WILL THERE BE ?

Not a lot ! Here' s an example from a part of a branch block dump although it is totally out of the scope to document branch blockdumps. A branch block refers to a leaf block or to another branch block (depends on the height of the index) the contents of a branch will vary depending on to which kind of block -leaf or branch- it refers.

Branch block dump

```

=====
header address 207251532=0xc5a684c
kdxcolev 2
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 4
kdxcosdc 0
kdxconro 47 >> seems to be the number of branch block entries (entries in this branch block)
kdxcofbo 122=0x7a
kdxcofeo 5673=0x1629
kdxcoavs 5551
kdxbrlmc 16836786=0x100e8b2 >> seems to be the rdba of first leaf block to which the branch block
refers (still to find out whether it can also be the first branch block of the level + 1 to which this branch
block refers)
kdxbrsno 0
kdxbrbksz 8060 >> seems to be the block size minus the overhead (O1+O2)
kdxbr2urrc 10
row#0[8009] dba: 16836942=0x100e94e
col 0; len 3; (3): 34 31 30
col 1; len 14; (14): 34 30 30 2f 30 32 32 2f 49 32 32 30 33 31
col 2; len 20; (20): 34 31 30 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
col 3; len 6; (6): 01 00 21 d6 00 16

...

```

Let us take a look on the number of branch blocks compared with the number of leaf blocks.

```
SQL> select lf_blks,br_blks,pct_used,height,lf_rows from index_stats;
```

```

LF_BKLS BR_BKLS PCT_USED HEIGHT LF_ROWS
-----
7251      49          80      3      899645

```



Oh yes there is also 1 root block. What will the ratio branch+root / leaf be ?

```
SQL> select round((49+1)*100/7251,2) " RATIO BRANCH/LEAF " from dual;
```

```

RATIO BRANCH/LEAF
-----
.69

```

	<p>Estimate size of B-tree Indexes</p> <hr/> <p>Version : 0.3</p>	
---	---	---

0.69% is that a lot ? Should we calculate the size, pctfree of them keeping in mind that we never can achieve exact science with the leafs ? To my opinion no. I tracked down that the formula I posted above gave me almost everytime a positive delta and I decided since this positive delta covers the branch + root extra bytes to leave it for leafs + branch + root. (or should I have said to "leaf" it)

5. CONCLUSION

In order to estimate the size of the indexes I will use

$[(A + vsize(B) + x + 1) * n \text{ indexed table rows}$

O

A = 6 bytes or

A = 10 bytes for non partitioned indexes on partitioned tables.

Is this exact science ? No **it is to be used as an estimation.** (1 + 1 = 2 and not 1,98 + branch + root = 1,99). I decided to do a set of tests. Please find back the results of different tests at the next page.

It seems the formula is reliable for single column indexes whereas it is a bit less accurate for composite indexes. It is also a bit less accurate when indexes are build in a 16K tablespace. Up to you to adjust the factor O to 0,75 or not.

The formula can very likely also be used for reverse key indexes whereas It CANNOT be used for

1. compressed indexes
2. function indexes (depends on the function)
3. domain indexes.
4. bitmap indexes

6. SPARSENESS OF LEAF BLOCKS

Since we can dump leaf blocks we can become aware of the number of index entries per leaf block. Assume we have a small index `IDX_GUID_OBJECT` on a table `T_OBJECT`, an index with only 233 leaf blocks. Right now we wonder how many index entries there are for each leaf block. Interesting to dump 233 leaf blocks ? Say no. Mr Jonathan Lewis has shown us how we can use the undocumented function⁷ `sys_op_lbid` in order to find out the density of the 233 leafs. Please feel free to look here http://www.jlcomp.demon.co.uk/index_efficiency.html. I became aware of this function by a tip from Mr Don Burleson. You can sign in here . <http://www.dba-oracle.com/> in order to receive this tips from him and from Mr Mike Ault.⁸

Back to work.

```
SQL> analyze index idx_guid_object validate structure;
Index analyzed.
```

```
SQL> select lf_blks from index_stats;
233
```

```
SQL> select /*+ cursor_sharing_exact dynamic_sampling(0) no_monitoring no_expand in
IDX_GUID_OBJECT) */ sys_op_lbid( 55192 , 'L', T_OBJECT.rowid) as block_id, count(*)
from MY_UTF.T_OBJECT group by sys_op_lbid( 55192 , 'L', T_OBJECT.rowid)
```



```
AAANeYAAFAAAPuOAAA      110
AAANeYAAFAAAPuPAAA      105
AAANeYAAFAAAPuQAAA      107
AAANeYAAFAAAPuRAAA      95
AAANeYAAFAAAPuSAAA      106
...
AAANeYAAFAAA6sHAAA      98
AAANeYAAFAAA6sIAAA      86
```

```
233 rows selected.
```

By using `sys_op_lbid` in the above way we can list the number of index entries per leaf block. If we are interested to know how many leaf blocks there are with n number of index entries we can use this one :

⁷ You may want to be prudent in a production db with `sys_op_lbid` since it is undocumented

⁸ I have in this document referred to several masters of who I learned a lot by reading their excellent white papers, forums, books : Mr Tim Gorman, Mr Thomas Kyte, Mr Jonathan Lewis, Mr Don Burleson, Mr Mike Ault. However no one of them have any responsibility regarding the correctness of this document.

	Estimate size of B-tree Indexes	
	Version : 0.3	



```
SQL> select rows_per_block, count(*) blocks from (select /*+ cursor_sharing_exact
dynamic_sampling(0) no_monitoring no_expand in IDX_GUID_OBJECT) */ sys_op_lbid( 55192
,'L',T_OBJECT.rowid) as block_id,count(*) as rows_per_block from SISII.T_OBJECT group by
sys_op_lbid( 55192 ,'L',T_OBJECT.rowid)) group by rows_per_block;
```

```

76      2
77      1
79      2
80      2
...
155     2
158     2
```

52 rows selected.

So we have 2 leaf blocks with 76 index entries, 1 with 77 entries and 2 with 158, ...
 By using sys_op_lbid in the above described ways we can maybe become aware
 whether it can **maybe** interesting to rebuild the index. (in a lot of cases it is not)
 Again, this is something I didn' t find myself. All the honour is for Mr Jonathan
 Lewis.

	Estimate size of B-tree Indexes	
	Version : 0.3	

Please feel free to mail me if you have comments both positive and negative. guy.lambrechts@telenet.be I offer a free Belgian Abbey Beer for you if you help me to improve the quality of my white paper.⁹

7. REFERENCES

“Online Operations on Indexes and Tables in Oracle 9i” : Metalink doc id 159063.

“Understanding Indexes” Mr Tim Gorman http://www.evdbt.com/2004_paper_549.doc

Oracle9i Space Management Demystified White Paper : Metalink doc id 247752.1

“Inside Oracle 9i Tablespace Management” Mr Don Burleson <http://www.dbazine.com/oracle/or-articles/burleson11>

“Interested Transaction List (ITL) Waits Demystified” Mr Arup Nanda <http://www.dbazine.com/oracle/or-articles/nanda3>

“Index Efficiency” Mr Jonathan Lewis http://www.jlcomp.demon.co.uk/index_efficiency.html

⁹ Belgium, country of Kim Clijsters and many, many good beer.

ESTIMATION BASED ON FORMULA

INDEX STATS INFO

INDEX_NAME	DATA_TYPE	avg(sum (col size)) in bytes	X	Num_rows	Calc. Size	Leaf Block Size	Branch Blocks	Index Stats Size (leaf+branch+root)
IX_EQ_ZONE	VARCHAR2	3	4	899645	17992900	8192	2122	6
IX_EQ_STRING12	CHAR(20)	20	4	899645	39841421	8192	4544	21
IX_EQ_ZONE_STRING12	VARCHAR2+CHAR	23	5	899645	44982250	8192	5112	26
PK_EQP_TOPO	NUMBER(8)	4	3	899645	17992900	8192	2138	4
PK_EQP_TOPO_ZONE	NUMBER(8)+VARCHAR	7	5	899645	24418936	8192	2839	6
PK_EQP_TOPO_STRING12	NUMBER(8)+CHAR	24	5	899645	46267457	8192	5242	9
IX_WO_ZONE	VARCHAR2	3	4	322121	6442420	8192	760	3
IX_WO_COSC	VARCHAR2	13,73	4	322121	11380075	8192	1307	6
IX_WO_ZONE_COSC	VARCHAR2+VARCHAR2	16,73	5	322121	13220766	8192	1507	7
PK_WIP_WO	NUMBER(8)	4	3	322121	6442420	8192	761	3
IX_EQ_ZONE	VARCHAR2	3	4	899645	17992900	16384	1045	3
IX_EQ_STRING12	CHAR(20)	20	4	899645	39841421	16384	2233	6
IX_EQ_ZONE_STRING12	VARCHAR2+CHAR	23	5	899645	44982250	16384	2513	7
PK_EQP_TOPO	NUMBER(8)	4	3	899645	17992900	16384	1052	1
PK_EQP_TOPO_ZONE	NUMBER(8)+VARCHAR	7	5	899645	24418936	16384	1398	3
PK_EQP_TOPO_STRING12	NUMBER(8)+CHAR	24	5	899645	46267457	16384	2583	3
IX_WO_ZONE	VARCHAR2	3	4	322121	6442420	16384	375	1
IX_WO_COSC	VARCHAR2	13,73	4	322121	11380075	16384	643	3
IX_WO_ZONE_COSC	VARCHAR2+VARCHAR2	16,73	5	322121	13220766	16384	743	3
PK_WIP_WO	NUMBER(8)	4	3	322121	6442420	16384	375	1
PK_T_OBJECT	NUMBER(10)	3,88	3	19873	394053	8192	44	1
IDX_GUID	VARCHAR2	38	4	23545	1648150	8192	183	1

Column Calc. Size is the calculated size (by the formula)

[(A + vsize(B) + x + 1) * n indexed table rows

O

A = 6 , O = 0,7

(A = 10 for non partitioned indexes on partitioned tables and for global partitioned indexes)

Column Index Stats Size is (leaf blocks+branch blocks+root block) * block size